DESIGN DOCUMENT

CSE 589 FALL 2016

PROJECT 2

ROUTING PROTOCOLS

NAME : CHAITANYA VEDURUPAKA
UBIT   : cvedurup
PERSON NUMBER : 50205782

In this project, I have implemented Distance Vector Protocol on top of 5 servers using UDP. The program supports the following user input commands

1. Update – Update the link cost
2. Display – Display <destination-server-ID> <next-hop-server-ID> <cost-of-path>
3. Step – Send routing update to neighbors right away
4 .Disable - Disable the link to given server.
5. Crash - Emulate a server crash. Close all connections on all links and exit
6. Packets - Display the number of distance vector packets this server has received.

In addition, it also sends routing updates in a regular interval to all the neighbors.

Programming Language Used : C++

SOURCE FILES : MainRouting.cpp , DistanceVector.cpp
HEADER FILES : MainRouting.h , DistanceVector.h , Defines.h

MainRouting.cpp - This is the Main source file. Defined **Routing** class here which is the main class which executes all the user input commands and also handles UDP connections with all the neighbors. This file also contains a class defined as **HelperFunctions** which is used to do small operations like string to int conversion , tokenizing etc.
DistanceVector.cpp – This file contains the implementation of distance vector. **DistanceVector** class is defined here which is used to compute the shortest distance to all the servers in the network using Bellman-Ford equation.
Defines.h – Contains macros used in project wide.

Data Structures used:

1. The data structure used to represent Routing Table is a 1D array **selfDV[5]** of Structures of the below type

```
struct DVTable
{
    int serverID;
    int nextHop;
    int cost;
};
```

This structure is declared at line 7, **DistanceVector.h** file.
The above array structure is filled every time we perform distance vector.

2.The data structure used to represent neighbor DV’s is a 2D array **neighborDV[5][5]** of Structures of the below type

```
struct DVTable
{
    int serverID;
    int nextHop;
    int cost;
};
```

This structure is declared at line 7, **DistanceVector.h** file.
The above 2d array structure is filled when we receive Distance vector updates from its neighbors.

3. The data structure used to represent all the server details in the network i.e server Id , port number , IP is 1D array **sList[5]** of the below type

```
struct ServerList
{
    char IP[16];
    int portNumber;
    int serverID;
};
```

This structure is declared at line 16, **MainRouting.h** file.

The above structure is filled after reading the topology file.

4. The structure used to represent all the neighbor details i.e cost and server Id is 1D array **nList[5]** of the below type

```
struct NeighborList
{
    int serverID;
    int cost;
};
```

This structure is declared at line 30, **MainRouting.h** file.
The above structure is filled after reading the topology file and also gets updated when update link operation is performed on itself or its neighbors.

5. The data structure used to represent the Update Message is an array of unsigned char bytes **unsigned char* buffMsg**. To fill up the update message, I have used the info from the above structures like **DVTable** structure and **ServerList** structure and filled the Update Message in byte format. The API used for this implementation is **GetFormatedMessage()** @Line 205 , MainRouting.cpp. The update message is filled byte by byte in an unsigned char buffer and sent to destination server. At the destination server , the Update message is read byte by byte and the corresponding Distance Vector is updated. The API used for this implementation is **ReadFormattedMessage()** @Line 143 , MainRouting.cpp.

6. The data structure used to represent the update link message which is sent to other neighbors whenever **UPDATE** command is executed is

```
struct UpdateFormat
{
   int format; → Used to identify if it is a Cost update message or not ,set to 0xFFFFFFFF
   int serverID;
   int cost;
};
```

This structure is declared at line 23, **MainRouting.h** file.
The above structure is filled after Update command command is executed.

<u>Execution and working of the program:</u>
The program can be executed using the below command :

```
./project2 -t <topology-file-name> -i <routing-update-interval>
```

During the start of the program, topology file is read and all the data is filled in the respective structures like **ServerList** structure and **NeighborList** structure and then the initial routing table is built using the available info . The API used for this is **ReadTopologyFile()** @Line 913 , MainRouting.cpp. Then the **Manager()** (@Line 678,MainRouting.cpp) API is called which is the main API that handles the complete program. In this API, we first create a UDP listening socket which is used to listen to the incoming datagram messages from other servers. Inside Manager() , System socket API **select()** is used to handle both STD input and incoming UDP messages. Here the select() API is used with the time interval argument as well , which is initially set to the given input interval. If the select() doesn't get hit before timeout, then periodic message API **PeriodicUpdateToNeighbors()** (@Line 318 , MainRouting.cpp) is called where it sends its distance vector to all the neighbors. If the select() gets hit before timeout (may be due to STDIN or UDP message from other servers), then the respective operation is performed and select() is called again but the time value in the last argument is the actual time left for the given time interval to be completed. The time left is computed using **gettimeofday()** (before calling select() and after select gets hit) implemented in the API **GetCurrentTimeInMS()** @Line 27 , MainRouting.cpp.

The Manager() calls the below API's for user input commands and update messages from neighbors.

<u>User Commands :</u>
Whenever user input is given , **ExecuteCommand()** (@Line 610 , MainRouting.cpp) is called which calls the respective API depending on the used command input. The following API's are called inside ExecuteCommand() depending on the command type :
- **ExecuteUpdate() :** To update cost between server Id and neighbor Id and then send an update message to neighbor. Later distance vector is computed.
- **ExecuteStep():** Triggers a periodic message immediately to its neighbors.
- **ExecutePackets():** Prints the number of distance vector packets received since the last time this API was called.
- **ExecuteDisable():** Disables connection with the neighbor server ID , by setting the neighbor cost to INF and stops sending/ receiving message from that neighbor. Later distance vector is computed.
- **ExecuteCrash():** Machine should close all connections and not recover from crash , it is emulated by exiting the program.
- **ExecuteDisplay():** Prints all the Destination server Id's , Next Hop Id's and their corresponding cost of paths.

<u>DV update / Cost update message:</u>
When the select() (inside Manager()) gets ready due to UDP message from neighbor , first the message is identified i.e whether it is a distance vector message or cost update message. This is done through the API **IdentifyRecvMsgType()** @Line 168 , MainRouting.cpp. If it is a distance vector message , the structure **neighborDV** is updated using the API **AddNeighborDVDetails() ,** @Line 94 , DistanceVector.cpp. If it is a cost update message , the neighbor's cost is updated. In both the cases the distance vector is computed immediately after the above operations are performed. The DV is computed in the API **ComputeDistanceVector()** @Line 103 , DistanceVector.cpp.

The below is the algorithm used to implement distance vector using Bellman-Ford equation:

```
//Find the DV to all the servers in the network
for(int i = 0 ; i < routingObj->numOfServers ; i++)
 {
    //Checking if the destination server is not itself
    if(routingObj->sList[i].serverID != routingObj->selfServerID)
    {
       for(int j = 0 ; j < routingObj->numOfServers; j++)
       {
          //Checking if it is a neighbor or not
          if(routingObj->nList[j].cost != 0 && routingObj->nList[j].cost != INF)
          {
             int nCost = routingObj->nList[j].cost;
             //Checking if it the neighbor has path to the destination server
             if(neighborDV[j][i].cost != INF)
             {
                //Bellman-Ford equation
                selfDV[i].cost = MINIMUM((nCost + neighborDV[j][i].cost),selfDV[i].cost);

                //Checking if the min cost is updated , if yes next hop is also updated
                if(selfDV[i].cost == nCost + neighborDV[j][i].cost)
                {
                   selfDV[i].nextHop = routingObj->nList[j].serverID;
                }
             }
          }
       }
    }

 }
```

Assumptions made while implementing this project:
1. The maximum number of servers is 5.
2. The user input commands are case insensitive.
3. INF is 65535
4. The server Id can be any positive integer.
5. The cost to self is zero and all other neighbors is non-zero or INF.
6. Server crash means exiting the program as the machine breaks connections with all the neighbors and cannot be recovered.

NOTE:
All the functions in the source code are commented and it can be referenced for more details regarding the API's and their implementation.